CIC111 — Análise e Projeto de Algoritmos II

Pedro H. D. B. Hokama

Originalmente elaborado pelos professores:
Cid C. de Souza
Cândida N. da Silva
Orlando Lee
Pedro J. de Rezende

Qualquer problema ou erro é de minha responsabilidade.

3 de maio de 2021

Árvore Geradora Mínima

Árvore Geradora Mínima: definição do problema

- Suponha que queiramos construir estradas para interligar n cidades, sendo que, a cada estrada entre duas cidades i e j que pode ser construída, há um custo de construção associado.
- *Como determinar eficientemente quais estradas devem ser construídas de forma a minimizar o custo total de interligação das cidades ?*
- Este problema pode ser modelado por um problema em grafos não orientados ponderados onde os vértices representam as cidades, as arestas representam as estradas que podem ser construídas e o peso de uma aresta representa o custo de construção da estrada.

P. Hokama, C. de Souza, C. Nunes, O. Lee, P. de Rezende CIC111 — Análise e Projeto de Algoritmos II — versão 2

Árvore Geradora Mínima: definição do problema

Nessa modelagem, o problema que queremos resolver é encontrar um **subgrafo gerador** (que contém todos os vértices do grafo original), **conexo** (para garantir a interligação de todas as cidades) e cuja soma dos custos de suas arestas seja a menor possível.

Mais formalmente, definimos o problema da seguinte forma:

Problema da Árvore Geradora Mínima:

Dado um grafo não orientado ponderado $G = (V, E)$, onde $w : E \to \mathbb{R}^+$ define os custos das arestas, encontrar um subgrafo gerador conexo T de G tal que, para todo subgrafo gerador conexo T' de G

$$\sum_{e \in T} w_e \leq \sum_{e \in T'} w_e.$$

Árvore Geradora Mínima

- Claramente, o problema só tem solução se G é conexo.
- **Portanto, a partir de agora, vamos supor que G é conexo**.
- Além disso, não é difícil ver que a solução para esse problema será sempre uma árvore: basta notar que T não conterá ciclos pois, caso contrário, poderíamos obter um outro subgrafo T', ainda conexo e com custo menor ou igual ao de T, removendo uma aresta do ciclo.
- Portanto, dizemos que este problema de otimização é o problema de encontrar a *Árvore Geradora Mínima* (AGM) em G.
- *Como projetamos um algoritmo para resolver esse problema ?*

Algoritmo para AGM

- Uma primeira abordagem exaustiva para solucionar o problema poderia ser enumerar todas as árvores geradoras do grafo, computar seus custos e retornar uma árvore de custo mínimo.
- No entanto, esse não é um bom algoritmo pois um grafo completo de n vértices possui um número **exponencial** (n^{n-2}) de árvores geradoras.
- Para obter um algoritmo eficiente (polinomial !) devemos então procurar alguma propriedade do problema que nos permita restringir o espaço de possíveis soluções a ser analisado.

Algoritmo genérico para AGMs

- Os dois algoritmos que serão vistos usam uma **estratégia gulosa**, diferindo apenas no modo em que esta é aplicada.
- A estratégia gulosa é resumida no *algoritmo genérico* mostrado a seguir, onde a AGM é construída aresta a aresta.
- A cada iteração do algoritmo é mantido um conjunto A de arestas que satisfaz à seguinte **invariante**:

> Antes de cada iteração, A é um subconjunto de arestas de *uma* AGM.

- Em cada iteração, determina-se ainda uma aresta (u, v) tal que $A \cup \{(u, v)\}$ que também satisfaz à invariante, i.e., está contido em alguma AGM. Tal aresta é dita ser **segura**.

Algoritmo genérico para AGMs

AGM-GENERICO(G, w)

```
1.  A ← ∅;
2.  enquanto A não forma uma árvore geradora faça
3.      encontre uma aresta segura (u, v);
4.      A ← A ∪ {(u, v)};
5.  retorne A.
```

- A parte mais *engenhosa* do algoritmo é encontrar a aresta **segura** na linha 3.
- Esta aresta deve existir pois, por hipótese, no laço das linhas 2–4, A é um subconjunto **próprio** de uma árvore geradora T. Logo existe uma aresta segura $(u, v) \in T - A$.

Árvores geradoras: reconhecendo arestas seguras

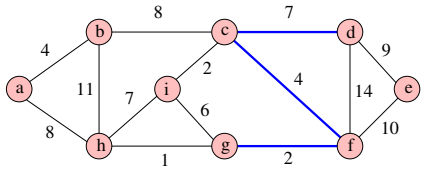
Notação:

Considere um grafo não orientado $G = (V, E)$ e tome $S \subseteq V$. O conjunto $V - S$ é denotado por $\overline{S}$.

Definição:

O *corte de arestas* de S, denotado por $\delta(S)$, é o conjunto de arestas de G com um extremo em S e outro em $\overline{S}$.

Exemplo: $S = \{a, b, c, g, h, i\}$.



Árvores geradoras: reconhecendo arestas seguras

Diz-se que um corte $\delta(S)$ **respeita** um conjunto de arestas A se $A \cap \delta(S)$ é vazio.

Diz-se que (u, v) é uma **aresta leve** de um corte $\delta(S)$ se $w_{uv} := \min_{(x,y)\in\delta(S)}\{w_{xy}\}$, i.e., (u, v) é a aresta de menor peso no corte $\delta(S)$.

Teorema 23.1: (Cormen 2ed)

Seja $G = (V, E)$ um grafo conexo não orientado ponderado nas arestas por uma função $w : E \to \mathbb{R}$. Seja A um subconjunto de E que está contido em alguma AGM de G, seja ainda $\delta(S)$ um corte de G que *respeita* A e (u, v) uma *aresta leve* de $\delta(S)$. Então, (u, v) é uma *aresta segura* para A.

Árvores geradoras: reconhecendo arestas seguras

O Teorema anterior deixa claro o funcionamento do algoritmo genérico para AGM. À medida que o algoritmo avança, o grafo induzido por A é acíclico (pois pertence à uma árvore). Ou seja $G_A = (V, A)$ é uma floresta, sendo que algumas árvores podem ter um único vértice. Cada aresta segura (u, v) conecta duas componentes distintas de G_A.

Inicialmente a floresta tem $|V|$ árvores formadas por vértices isolados. Cada iteração reduz o número de componentes da floresta de uma unidade. Portanto, após a inclusão de $|V| - 1$ arestas seguras, o algoritmo termina com uma única árvore.

Árvores geradoras: reconhecendo arestas seguras

Corolário 23.2: (Cormen 2ed)

Seja $G = (V, E)$ um grafo conexo não orientado ponderado nas arestas por uma função $w : E \to \mathbb{R}$. Seja A um subconjunto de E que está contido em alguma AGM de G, e seja $C = (V_C, E_C)$ uma componente conexa (árvore) da floresta $G_A = (V, A)$. Se (u, v) é uma *aresta leve* de $\delta(C)$, então (u, v) é **segura** para A.

Os algoritmos de Prim e de Kruskal

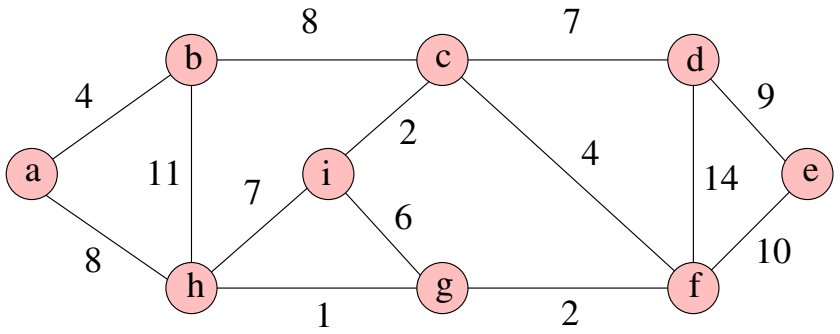
- Os algoritmos de Prim e de Kruskal especializam o algoritmo genérico para AGM visto anteriormente, fazendo uso Corolário 23.3.
- No *algoritmo de Prim*, o conjunto de arestas A é sempre uma **árvore**. A *aresta segura* adicionada a A é sempre uma *aresta leve* do corte $\delta(C)$, onde C é o conjunto de vértices que são extremidades de arestas em A.
- No *algoritmo de Kruskal*, o conjunto de arestas A é uma **floresta**. A *aresta segura* adicionada a A é sempre a aresta de menor peso dentre todas as arestas ligando dois vértices de componentes distintas de G_A.

O algoritmo de Prim

PRIM(G, w, r)

```
▷ r é um vértice escolhido para ser raiz da AGM
  Q é o conjunto de vértices a incluir na árvore
  dist[x]: peso da aresta mais leve ligando x a componente de r
1.  dist[r] ← 0;          Q ← V;
2.  para todo v ∈ V − {r} faça dist[v] ← ∞;
3.  pred[r] ← nulo;
4.  A ← {};               ▷ inicializa o conjunto de arestas da AGM
5.  W ← 0;                ▷ inicializa o peso da AGM
6.  enquanto (Q não for vazio) faça
7.      Remover de Q o vértice u com o menor valor em dist;
8.      W ← W + dist[u];
9.      se pred[u] ≠ nulo, A ← A ∪ {(pred[u], u)};
        ▷ atualiza o valor de dist[.] para vértices adjacentes a u
10.     para todo v ∈ Adj[u] faça
11.         se (v ∈ Q) e (dist[v] > w[u, v]) então
12              { dist[v] ← w[u, v];   pred[v] ← u; }
13. retorne (A,W).
```


O algoritmo de Prim: exemplo



O algoritmo de Prim: corretude

O algoritmo de Prim mantém a seguinte invariante, a qual é válida antes de cada execução do laço das linhas 6 a 12:

1. $A = \{(v, \texttt{pred}[v]) : v \in V - \{r\} - Q\}$.
2. Os vértices já colocados na AGM são aqueles em $V - Q$.
3. Para todos os vértices em Q, se $\texttt{pred}[v] \neq$ nulo, então $\texttt{dist}[v] < \infty$ e $\texttt{dist}[v]$ é o valor da aresta leve $(v, \texttt{pred}[v])$ que conecta v a algum vértice já pertencente a AGM.

O algoritmo de Prim: complexidade

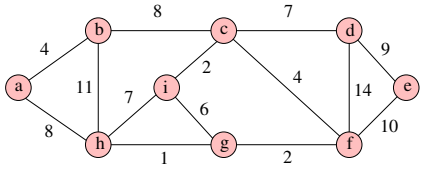
- A complexidade depende de como o conjunto Q é *implementado*.
- Se Q for implementado como um vetor simples (Q poderia estar representado pelo próprio vetor `dist`), a complexidade do algoritmo será $O(|V|^2 + |E|) \equiv O(|V|^2)$.
- Se Q for implementado como um heap (Q poderia estar representado pelo próprio vetor `dist`), a complexidade do algoritmo será $O((|V| + |E|) \log |V|) \equiv O(|E| \log |V|)$.
- Portanto, para grafos *densos* ($|E| \in O(|V|^2)$), a melhor alternativa é implementar Q como um vetor simples, enquanto que para grafos *esparsos* ($|E| \in O(|V|)$), deve-se optar pela implementação de Q como uma fila de prioridades.
- Note que, com esta última opção, o algoritmo de Prim assemelha-se muito a uma busca em largura. A diferença entre os algoritmos fica praticamente restrita à troca de uma fila simples por uma fila de prioridades !

O algoritmo de Kruskal para AGM

Principais passos do algoritmo de Kruskal:

1. Ordenar as arestas em ordem **não** decrescente de peso e inicializar A como estando vazio.
2. Seja (u, v) a próxima aresta na ordem **não** decrescente de peso. Se ela formar um ciclo com as arestas de A, então ela é **rejeitada**. Caso contrário, ela é **aceita** e faz-se $A = A \cup \{(u, v)\}$.
3. Repetir o passo anterior até que $|V| - 1$ arestas tenham sido **aceitas**.

Exemplo:



O algoritmo de Kruskal para AGM

- O algoritmo de Kruskal é baseado diretamente no *algoritmo genérico* discutido anteriormente.
- Note que quando uma aresta (u, v) é aceita, as componentes C_1 e C_2 contendo u e v, respectivamente, são *distintas*. É fácil ver que (u, v) é uma *aresta leve* para $\delta(C_1)$ (ou $\delta(C_2)$). Portanto, (u, v) satisfaz às condições do Corolário 23.3, o que garante a corretude do algoritmo.
- Em uma iteração qualquer, a solução parcial corrente é a floresta composta por todos vértices do grafo e as arestas em A.
- Para implementar o algoritmo de Kruskal, precisamos encontrar uma maneira eficiente de *manter as componentes* da floresta $G_A = (V, A)$.

O algoritmo de Kruskal para AGM

- Em particular, as componentes devem ser armazenadas de modo que duas operações sejam feitas muito rapidamente:
  - dado um vértice u, **encontrar** a componente contendo u;
  - dados dois vértices u e v em componentes distintas C e C', **unir** C e C' em uma única componente.
- para prosseguir com esta discussão, vamos apresentar um pseudo-código do algoritmo de Kruskal.
- Neste pseudo-código, denota-se por $a[u]$ a componente (árvore) de $G_A = (V, A)$ que contém o vértice u na iteração corrente.

Algoritmo de Kruskal: pseudo-código

KRUSKAL(G, w)

```
1.  W ← 0;   A ← ∅;   ▷ inicializações
    ▷ Iniciar G_A com |V| árvores com um vértice cada
2.  para todo v ∈ V faça a[v] ← {v};   ▷ a[v] é identificado com v
    ▷ lista de arestas em ordem não decrescente de peso
3.  L ← ordene(E,w);
4.  k ← 0;          ▷ conta arestas aceitas
5.  enquanto k ≠ |V| − 1 faça
6.      remove(L, (u, v));   ▷ tomar primeira aresta em L
        ▷ acha componentes de u e v
7.      a[u] ← encontrar(u);   a[v] ← encontrar(v);
8.      se a[u] ≠ a[v] então ▷ aceita (u, v) se não forma ciclo com
A
9.          A ← A ∪ {(u, v)};
10.         W ← W + w(u, v);
11.         k ← k + 1;
12.         unir(a[u], a[v]);   ▷ unir componentes
13. retorne (A, W).
```


Algoritmo de Kruskal: complexidade

- Supor que as componentes de G_A são mantidas de modo que as operações de **encontrar** na linha 7 e **unir** na linha 12 sejam feitas com complexidade $O(f(|V|))$ e $O(g(|V|))$, respectivamente.
- A ordenação da linha 3 tem complexidade $O(|E| \log |E|)$ e domina a complexidade das demais operações das inicializações das linhas de 1 a 4.
- O laço das linhas 5–12 será executado $O(|E|)$ no pior caso. Logo, a complexidade total das linhas 6 e 7 será $O(|E|.f(|V|)$.
- As linhas de 9 a 12 serão executados $|V| - 1$ vezes no total (por quê ?). Assim, a complexidade total de execução destas linhas será $O(|V|.g(|V|)$.
- A complexidade do algoritmo de Kruskal será então

$$O(|E| \log |E| + |E|.f(|V|) + |V|.g(|V|))$$

Fica claro que necessitamos de uma estrutura de dados que permita *manipular* eficientemente as componentes de G_A.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

- Note que os conjuntos de vértices das componentes de G_A formam uma *coleção de conjuntos disjuntos* de V. Sobre estes conjuntos é executada uma seqüência de operações **encontrar** e **unir**.
- A necessidade de representação e manipulação eficiente de coleção de conjuntos disjuntos sob estas operações ocorre em áreas diversas como construção de compiladores e problemas combinatórios, como é o caso da AGM.
- Estruturas de dados simples como vetores contendo informação sobre qual a componente contendo cada elemento (vértice) realizam a operação **encontrar** em $O(1)$ mas, no pior caso, consomem um tempo $O(|V|)$ para realizar uma operação de **união**.
- A alternativa é o uso de estruturas ligadas do tipo **árvore**

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

Cuidado ! Não confunda a estrutura de dados árvore que estaremos usando para armazenar as componentes de G_A com as árvores da floresta G_A.

Ou seja, nesta estrutura os registros estão ligados numa estrutura tipo árvore **cujos apontadores ligando registros de vértices distintos não necessariamente correspondem a uma aresta de** A (pode ser inclusive que nem exista uma aresta com estas mesmas extremidades no grafo).

Para evitar confusão o termo *componente* será usado na discussão que se segue para designar vértices em uma mesma árvore de G_A. O termo *árvore* denotará uma parte da estrutura de dados que representa uma componente de G_A.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

Na discussão que se segue, supomos que, no início, seja criado um **registro** para cada vértice u de V e que o endereço deste registro esteja armazenado em alguma variável de modo que possa ser acessado a qualquer momento do algoritmo.

Além disso, cada registro r da estrutura terá dois campos obrigatórios (`M`) e um terceiro campo opcional (`O`):

- `rot (M)`: contendo o rótulo do vértice que o originou;
- `prx (M)`: apontador ligando registros de uma mesma árvore;
- `num (O)`: número de registros da estrutura de dados tais que, seguindo o campo `prx` até que este seja nulo, chega-se ao registro r.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

encontrar(u):

▷ retorna o apontador para o registro que representa a componente onde se encontra o vértice u

1. $p \leftarrow$ endereço do registro correspondente a u;
2. **enquanto** ($p\hat{}$.`prx` $\neq$ nulo) **faça**
3. $\quad p \leftarrow p\hat{}$.`prx`;
4. **retorne** p.

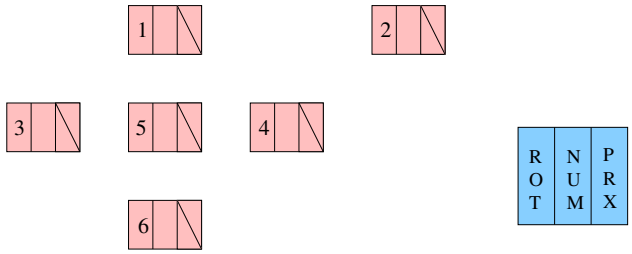
unir(p,q): versão inicial

▷ **Entrada**: dois apontadores para registros correspondentes a componentes distintas de G_A

▷ **Saída**: apontador para o registro correspondente ao primeiro parâmetro da entrada

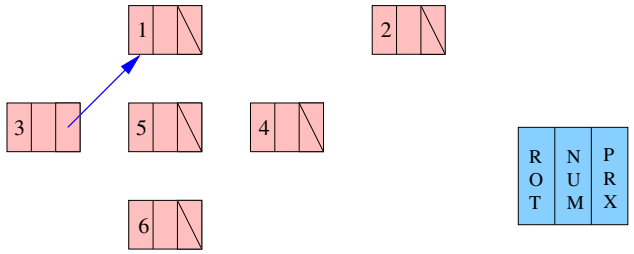
1. $q\hat{}$.`prx` $\leftarrow p$
2. **retorne** p.

Conjuntos disjuntos: exemplo



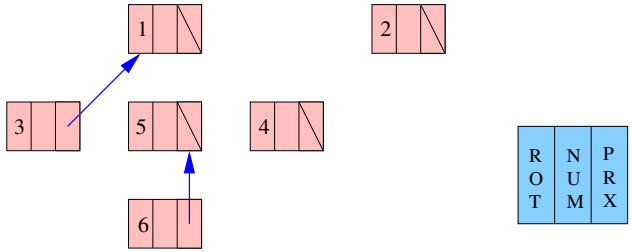
unir(a[1], a[3])

Conjuntos disjuntos: exemplo



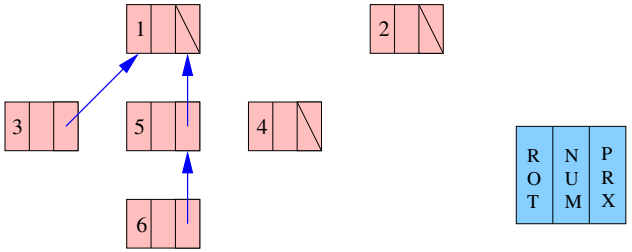
unir(a[5], a[6])

Conjuntos disjuntos: exemplo



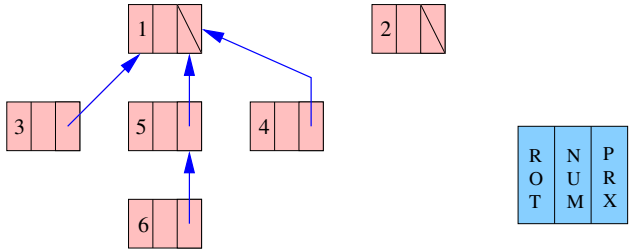
unir($a[1], a[6]$)

Conjuntos disjuntos: exemplo



unir($a[5], a[4]$)

Conjuntos disjuntos: exemplo



Conjuntos disjuntos: complexidade das operações

Complexidades:

Denote-se por T_x a árvore contendo o registro do vértice x e h_x e n_x respectivamente sua altura e seu número de elementos. No pior caso teríamos as complexidades $O(h_u)$ para **encontrar(u)** e $O(1)$ para **unir(p,q)**.

Dificuldade:

Uma das árvores pode se transformar em um longo *caminho*. Neste caso, uma operação envolvendo esta árvore teria uma complexidade de pior caso muito alta ($O(|V|)$).

Alternativa:

Construir *árvores balanceadas* colocando no campo `num` do registro r, o número de registros na sub-árvore "abaixo" de r.

Conjuntos disjuntos: complexidade das operações

unir(p,q): versão melhorada

▷ **Entrada**: dois apontadores para registros correspondentes a componentes distintas de G_A

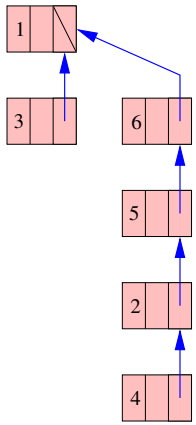
▷ **Saída**: apontador para o registro com maior valor no campo `num`

```
1.  se  q^.num ≤ p^.num então
2.      q^.prx ← p;
3.      p^.num ← p^.num + q^.num;
4.      retorne p.
5.  se não
6.      p^.prx ← q;
7.      q^.num ← q^.num + p^.num;
8.      retorne q.
```

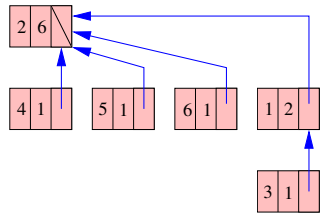

Conjuntos disjuntos: complexidade das operações

Seqüência de operações: unir(a[1], a[3]), unir(a[2], a[4]), unir(a[5], a[2]), unir(a[6], a[5]) e unir(a[1], a[6]).

união não-balanceda:



união balanceda:



Conjuntos disjuntos: complexidade das operações

Teorema:

Considere uma seqüência $s_0, s_1, \ldots, s_m$ de operações sobre uma estrutura de dados representando conjuntos disjuntos. A operação s_0 simplesmente cria os registros representando os conjuntos unitários formados por cada um dos elementos. Em $s_1, \ldots, s_m$ todas as operações são do tipo **encontrar** ou **unir**, sendo esta última feita de modo *balanceada*.

Seja x um registro qualquer da estrutura e T_x a árvore que armazena x. Após a operação s_m, T_x é uma **árvore balanceada**.

Conjuntos disjuntos: complexidade das operações

Prova: seja n_x o número de registros em T_x e h_x a sua altura. Deve se mostrar que $2^{h_x - 1} \leq n_x$ e, consequentemente, fica provado que $h_x \in O(\log n_x)$.

A prova é feita por indução no número de operações. Na base, o resultado é trivialmente verdadeiro para s_0 já que toda árvore tem um único elemento.

Suponha que o resultado é verdadeiro até o término de s_{m-1}. Deve-se analisar dois casos: *(i)* s_m é uma operação **encontrar(u)** e *(ii)* s_m é uma operação **unir(v, w)**.

O caso *(i)* é trivial pois nenhuma árvore é alterada. No caso *(ii)*, sem perda de generalidade, vamos supor que $n_v \leq n_w$.

Conjuntos disjuntos: complexidade das operações

Prova (cont.):
Sejam T_v e T_w as árvores contendo os registros dos vértices v e w, respectivamente. Seja T' a árvore resultante da operação **unir(v, w)** com altura e número de registros dados por h' e n', respectivamente. Note que $n' = n_v + n_w$ e que deve ser provado que $2^{h'-1} \leq n'$.

Claramente $h' = \max\{1 + h_v, h_w\}$. Tem-se que

$$\begin{aligned} 2^{h'-1} &= 2^{\max\{1+h_v,h_w\}-1} = \max\{2^{h_v-1+1}, 2^{h_w-1}\} \\ &\leq \max\{2n_v, n_w\} \qquad \text{(pela H.I.)}. \end{aligned}$$

Como $n' = n_v + n_w$ e $n_v \leq n_w$, então $2n_v \leq n'$ e $n_w \leq n'$.
Logo, $2^{h'-1} \leq n'$. $\square$

Algoritmo de Kruskal: complexidade

- Vimos anteriormente que se as operações de **encontrar** e **unir** fossem feitas com complexidades $O(f(|V|))$ e $O(g(|V|))$, respectivamente, a complexidade do algoritmo de Kruskal seria dada por $O(|E| \log |E| + |E|.f(|V|) + |V|.g(|V|))$.
- O resultado do Teorema garante que usando a **união por tamanho** as árvore de registros na estrutura de dados tem altura limitada a $O(\log |V|)$.
- Portanto, neste caso, a complexidade do algoritmo de Kruskal é dada por $O(|E| \log |E| + |E| \log |V| + |V|)$ ou, como estamos supondo que o grafo é conexo ($|E| \in \Omega(|V|)$), a complexidade é $O(|E| \log |E|) = O(|E| \log |V|)$.
- Melhorias na complexidade ainda podem ser alcançadas usando a técnica de **compressão de caminhos** ao se executar uma operação **encontrar**.

Algoritmo de Kruskal: compressão de caminhos

Na técnica de compressão de caminhos, todo registro visitado durante uma operação **encontrar** que termina em um registro q tem seu campo `prx` alterado de modo a apontar para q.

encontrar(u): adaptada para compressão de caminhos

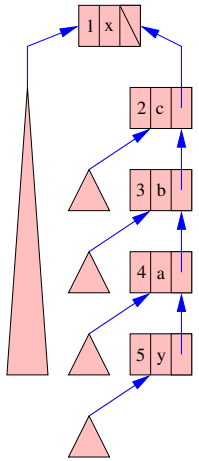
```
1. p ← endereço do registro correspondente a u;
2. q ← p;    ▷ guardará o último registro da busca
3. enquanto q^.prx ≠ nulo faça q ← q^.prx;
   ▷ refaz a busca fazendo todos registros apontarem para q
4. enquanto p^.prx ≠ q faça
5.     r ← p^.prx;
6.     p^.prx ← q;
7.     p ← r;
8. retorne q.
```

Nota: no algoritmo acima, a informação do campo `num` só é correta para o registro "raiz" da árvore.

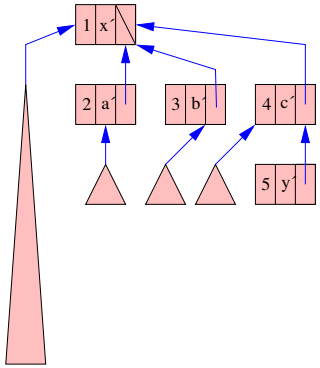
Compressão de caminhos: exemplo

encontrar(4):

Antes:



Depois:



Ajustando o valor de `num`: $x' = x$, $a' = a - y$, $b' = b - a$, $c' = c - b$, $y' = y$

Compressão de caminhos: complexidade

- Pode ser provado que, usando compressão de caminhos, a complexidade de se realizar $|V| - 1$ operações **unir** e $|E|$ operações **encontrar** tem complexidade dada por $O(|V| + |E|\alpha(|V| + |E|, |V|))$.
- Para compreender este resultado, é necessário conhecer a função de Ackerman A e o seu inverso α definidas por:

$$A(p,q) = \begin{cases} 2^q & p = 1,\ q \geq 1 \\ A(p-1,2) & p \geq 2,\ q = 1 \\ A(p-1, A(p,q-1)) & p \geq 2,\ q \geq 2 \end{cases}$$

$$\alpha(m,n) = \min\{p \geq 1 : A(p, \lfloor m/n \rfloor) > \log n\},\ m \geq n.$$

- É fácil ver que a função A tem crescimento muito rápido. Por exemplo:

$$A(2,1) = A(1,2) = 2^2$$

$$A(2,x) = A(1, A(2, x-1)) = 2^{A(2,x-1)} = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{\text{(x+1) vezes}}$$

Compressão de caminhos: complexidade

- A função α satisfaz $\alpha(m,n) \leq 4$ para *todos efeitos práticos*.
- Ou seja, a complexidade $O(|V| + |E|\alpha(|V| + |E|, |V|))$, embora não seja linear, se comporta como tal para valores práticos de $|V|$ e $|E|$.
- Note que $A(4,1) = A(2,16)$, ou seja, 17 potências sucessivas de 2. Então, se $m \approx n$, na expressão $\alpha(m,n)$, pode-se aproximar o valor de p para 4 em qualquer aplicação prática, já que

$$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{\text{17 vezes}} > \log n,$$

para qualquer valor razoável de n.